A Key Technology Evaluation Case Study: Applying a New Middleware Architecture on the Enterprise Scale

Michael Goedicke and Uwe Zdun

Specification of Software Systems, University of Essen, Germany,
{goedicke|uzdun}@informatik.uni-essen.de

Abstract Decisions for key technologies, like middleware, for large scale projects are hard, because the impact and relevance of key technologies go beyond their core technological field. E.g., object-oriented middleware has its core in realizing distributed object calls. But choosing a technology and product also implies to adopt its services, tools, software architectures, object and component paradigms, etc. Moreover, legacy applications and several other key technologies have to be integrated. And since no middleware product serves all requirements in the enterprise context, various middleware products have to be integrated, too. Another key problem of middleware evaluation is, that often the studies have to be performed very early in a project. In this paper we try to tackle these problems and describe how we can communicate the outcomes – which come from a technical viewpoint – to the management and other non-experts in the technological field.

1 Key Technology Evaluation Case Studies

Early key technology evaluations are a case study type that we have performed several times in different business setups for different companies. They aim at the very early investigation of key technologies, like communication infrastructure, database management systems, or programming languages, for large-scale, business-critical projects from a technical viewpoint. The projects were focussed on re-engineering of large existing systems with numerous applications, though many similar studies are performed for development projects of new software systems. In such projects management normally wants to decide which key technologies the project will use, before the project is actually launched, in order to estimate the savings/costs caused by the technology.

With the term "key technology" we generally refer to business-critical technologies, that drive companies to launch evaluation studies in early phases of (large-scale) projects. With other words: technologies that lead to high costs, if they fail to satisfy the company's expectations. Examples of key technologies could be communication infrastructure, database management, programming language, operating system, etc. technologies. It heavily depends on the nature of the project, whether one of these technologies is a key technology for the project or not.

In this paper we will concentrate on a study, which aims at the re-organization of the information systems of a large German enterprise with its core business in the field of logistics. The case study is embedded in a process that includes modeling the business processes, designing appropriate conceptual models, strategic decisions for used technologies, and specification of platforms. The presented case study had the task to identify and exemplify suitable middleware solutions for the information system base-line architecture of the enterprise. The great challenge of this study was, that it had to analyze key aspects of technology, before all application areas have named their actual requirements. The enterprise was confronted with the problem that a huge number of applications were developed independently by the various departments. But these applications had to work in concert to a certain degree in order to allow the departments to flexibly interoperate and to exchange their respective information.

Newer key technology trends, which have become mainstream recently, like distributed object systems, normally promise a lot, but also imply a set of risks. Since the company had not experts in (all) the new key technology areas, it was hard for the management to estimate which new technologies are valuable/necessary for the company. Therefore, a middleware evaluation study was launched during early requirements analysis. We believe that this situation is recurring for different key technologies in many companies of various size and in various business fields.

1.1 Software Architectural Integration

Often departments have had the freedom for a “programming in the wild” with no clear architectural concept for integration. This freedom helps to rapidly develop applications from the scratch. But when the number and complexity of applications rises, maintenance and integration of application becomes more and more difficult. In such a diverse field, like communication infrastructure, a lacking integration concept means not only to run into problems with integrating communication technologies, but also in integrating the various programming languages, object models, services, access variants to shared resources, etc. Many organizations, like the enterprise in this case study, react by creating an external department that should impose standards over information system development. Often these standards tend to be loaded with severe constraints. Therefore, often the standards are either ignored by the developers or they lead to monolithic systems that are hard to maintain.

These problems are similar in many companies, but they are more pressing in the enterprise context, than in small companies. There are several reasons. I.e., the enterprise has more different applications and technologies that have to be integrated. Applications have to be deployed to more hosts and middleware products have to be purchased earlier. Departments that impose standards have more likely limited personal contact to all affected developers and, therefore, developers are less involved in key technological decisions.

From our experience, an early technological study should give guideline and communication assistance for technological decisions to the concrete application's software architect rather than to make premature decisions in her/his place. Therefore, in this paper we will try to show a way to avoid such rather random impositions. Nevertheless, the paper comes to concrete results from managerial and technological viewpoints for the integration of object-oriented middleware in a very early project phase.

1.2 Roadmap

So far we have sketched a problem field, which seems on the first glance to have no clear solution. Management demands for a clear basis for decision and for an architectural perspective, but the field is much too diverse to provide a simple answer. However, simplicity and transparency of decisions are of central importance. Both the involved managers and developers have to be able to understand the decisions and the reasonings behind them directly. The acceptance of decisions relies on the solution's ability to cover the technical realm of the application in focus. In early evaluation studies this is very hard, because the application specifications do not exist when the study is launched. In enterprise-scale studies it is even harder, because a wide range of applications has to be covered.

Another problem we cannot ignore is subjectiveness. If people have to decide for key technologies personal experiences, predilections, opinions, and prejudices come into the decision. Our experience is that no pseudo-objective decision process can change this as long as not enough "hard" criteria can be found.

Our approach relies on the simple idea – which is not so simple in its realization – to use the right tool for a given task and then seek for technical solution for integration of these tools. However, the goal to find a company-wide integrated solution may cause damages that outweigh the benefits by far. In contrast to the concrete applications, the technological requirements for integration are quite concrete even in early stages, because the technologies themselves are already existing. We will see, that in this domain we face recurring problems, like object system integration, finding of a suitable component concept, bridging between technologies, etc. It is important that these integration decisions are performed at a detailed technological level. Otherwise the evaluation case study bears little to no technological substance and is unlikely to be accepted by concrete application developers.

In detail, we firstly have discussed and communicated the technological field, the available technology types, and the available products with the stakeholders.

The outcome was a document describing all these aspects in detail and a taxonomy of the middleware technologies/products. Thus we have come to consensus on all these aspects covering the objective *and* the subjective knowledge about the middleware technologies/products. We have created a "common language" for further discussion. These issues are discussed in the Sections 2 and 3.

With this "common language" we have then begun to discuss concrete application scenarios, that the stakeholders characterize as typical for the company. From our experience the outcome is nearly always a diverse set of technologies. Therefore, two questions arise directly: How do we integrate the different applications build with different technologies and how do we find the best technology from the taxonomy for a concrete application?

We provided a concrete technological concept for the integration task, which we introduce briefly in Section 4. As a guidance for concrete application decision we use a scenario-based process. In order to illustrate this process, we exemplify it with the typical application scenarios, which we had used earlier throughout the discussions. In Section 5 we present one such example of a letter distribution center information system.

2 Middleware

At the enterprise level the expectations are high and are often beyond the functionalities of existing products. At an early stage of a project it is hard to determine which technologies meet the requirements of an enterprise. Software development depends on several changing technologies. One of the most complex technology areas – faced in today's technology decision processes – is middleware [3]. In the context of this paper we see a middleware as following:

> *A middleware extends the platform with a framework comprising components, services, and tools for the development of distributed applications. It aims at the integration, the effective development, and the flexible extensibility of the business applications.*

Middleware comprises several different technologies provided by different vendors, which conform to a different extent to a great number of partially overlapping standards. Middleware abstracts from the network layer and from direct network access. Applications access networking functionalities through a well-defined interface of the middleware and communicate virtually on top of the middleware among each other. The middleware implements the details of network access (as illustrated in Figure 1). It provides a software layer between operating system functionalities and applications [17]. Thus it is often seen as an extension of the traditional notion of the platform.

Choosing a key technology, like middleware, has severe impact on the *software architecture* of the enterprise's information systems. The software architecture

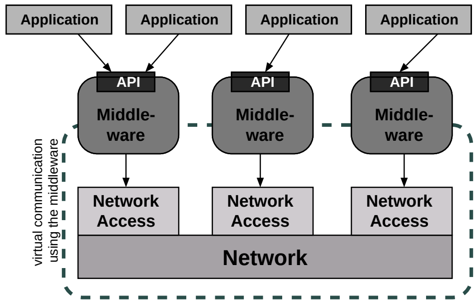


Figure 1. Middleware

consists of software components and the relations among them. Therefore, the component (and object) model of the middleware has to be integrated, adapted, and/or adopted to the component models used throughout design and implementation. Limitations of the middleware's component model can restrict the expression power accessible in design/implementation. But the middleware technology can also introduce new concepts and architectural means not accessible in the used design/implementation languages. E.g., the new CORBA 3 component model introduces a distinct component model into languages that offer none (like C) or Orbix Filters [9] introduce interception techniques into non-interceptible languages (such as Java).

Different middleware products (and technologies) offer a different spectrum of *services*. Services are central for the usage of middleware in the enterprise. Generally each service can also be developed by the enterprise itself (or by a third party). But standardized services allow the applications to be developed faster and more cost effective and they provide a higher interoperability among applications, that need such a service. Important service areas are messaging service, transaction service, security service, and naming (directory) service.

Even the *software development processes* are strongly influenced by middleware technologies. E.g., a clear component/interface model enables software development in separate teams. Therefore development with a component model enforces another development process than the development of a monolithic piece of software. For these reasons a positive impact of software architecture, a suitable set of services and tools, and the ability to enforce a promising development process are central requirements for a middleware technology. Besides these central requirements other non-technical requirements have to be considered, like costs of licenses, education of developers and designers, vendor politics, standards, existing education of the SW developers, designers, etc.

3 Technologies and Taxonomy

In the preceding sections we have tried to define middleware and to name important aspects when choosing a middleware for an enterprise. These aspects are chosen in order to be able to explain the impact of middleware technology onto distinct areas of software development. We have used these aspects to develop together with the stakeholders in the enterprise a common framework in order to characterize and compare different products in different categories. The specific framework was:

- Interoperability:
  - (Standardized) communication over the network,
  - Support for various programming languages,
  - Support for various platforms (i.e., platform independence).
  - Integrated component model.
- Services:
  - Messaging service,
  - Transaction service,
  - Security service,
  - Naming (directory) service.
- Scalability.
- Performance.
- Standardization.
- Marketablility of the products.

Note, that even this quite generic evaluation framework is subjective and company-specific. In other companies some aspects would probably more or less prominent, probably different service areas would have been chosen as important, etc. Every enterprise has to build its own framework and taxonomy when making an important technological decision. For different settings other central aspects, as for instance application deployment, have to be considered as well. In general the enterprise's set of important quality attributes has to be mapped onto a set of criteria that distinguishes the technologies clearly (e.g., when deciding about database technology the same technique may be applied).

The mapping can only be found through ongoing discussion with the stakeholders throughout the study, since they know their business case the best. Afterwards the enterprise can use the resulting taxonomy for their concrete applications. The reason for using a taxonomy is transportation of knowledge to the system's stakeholders, like management, developers, customers, etc. A middleware expert will know without building a taxonomy when to apply CORBA and when to use an RPC approach. But with a taxonomy it is easy to communicate such decisions, if the taxonomy is, on the one hand, based upon the central quality attributes of the enterprise and does, on the other hand, represent characteristic properties of the technologies.

3.1 Middleware Technologies

After we agreed with the stakeholders in the enterprise upon the framework that is able to express the relevant technologies and captures the important quality attributes of the company, we decided which technologies/products had to be investigated. These were:

- RPC Mechanisms, like: Sun RPC, OSF's DCE.
- Distributed Object Systems (based on the RPC principle):
  - CORBA - ORBs, like Orbix, Visibroker, TAO.
  - DCOM,
  - Enterprise Java Beans (EJB) and Java RMI,
- Application Server:
  - EJB-based servers, like WebLogic, Oracle Application Server, WebSphere.
  - Scripted web-servers, like Vignette V/5, AOLServer, Ajuba2, WebShell, Zope.
- Transaction Processing (TP) Monitors, like: Tuxedo, Encina.
- Message Oriented Middleware (MOM), like: MQSeries, Tibco.
- Mobile Code Systems (MCS), like: Aglets, Voyager, Telescript, Jini.

Finally we gave a two/three pages description of each technology and each product. The descriptions consist of a brief description, an illustrating figure, and an textual evaluation of each item of the taxonomy framework. Here, we just give a heavily abbreviated discussion of the CORBA technology as an example of the style of presentation (see Figure 2).

3.2 Results of the Assessments

The result of our assessments was a rather technical evaluation of the named middleware technologies and the different products in the various technological fields. Note, that these findings are *not* a generalizable view on middleware technology, which could be converted without change to another company. Instead these finding are strongly influenced by the process of discussing a suitable middleware in the realm of the actual company. The assessments reflect a lot of objective knowledge about middleware, but also a lot of subjective aspects, such as personal predilections, special experiences in the company, company politics, etc.

In summary, all participants in the discussion came to the opinion, that all technologies provide enhancements in the fields of the other technologies. TP monitors have their strength in database connection/transaction management. Application servers are superior in representing different business logics (like an additional web representation). But both technologies have only limited capabilities in other domains. Therefore – in the enterprise context – they should not be used as a sole middleware solution, if a part of the application (or even an expected future change) requires additional functionalities. Mobile code technologies can be superior to all other technologies in certain applications (e.g., when high configuration/customization needs on the server side are paired with low network bandwidth), but no marketable products are existing for the enterprise scale with all required middleware services.

That means for these three technologies we have hard (or better: merely objective) criteria for several applications whether they should be used or not. But even for these technologies it is most often not obvious at the first sight if the application benefits from one of these technologies more than from another. With all other technologies the decision for a technology is even harder. With other words: at the enterprise scale and under consideration of the diversity in the middleware field, there can be no a priori decision for one specific product that suits all application needs.

This point is a key problem of this work: How can we – as technical consultants – deal with a situation, when a managerial decision requires a foundation, but the technical field is so unwieldy or complex or entangled, that an objective statement is merely impossible. On the other hand, the manager requires a basis for decisions, so this statement alone is not a sufficient answer for her/him. Moreover, in any such complex decision field, we have to deal with a lot of subjective opinions and prejudices.

	RPC	CORBA	DCOM	EJB	AS	TP	MOM	MCS
Interoperability:								
Network communication	+	+ +	+	+	+	+	+	+
Programming language independence	-	+ +	+	- -	-	-	o	o
Platform independence	o	+ +	- -	+	+	+	+	+
Integrated component model	-	+	+	+	+	-	-	+
Services:								
Messaging services	-	o	-	o	o	-	+ +	+
Transaction services	o	+	o	o	o	+ +	-	- -
Security services	o	o	o	-	o	+	-	+
Naming (directory) services	o	o	o	o	o	o	-	-
Scalability	- -	+	-	+	+	+ +	+	o
Performance	o	+	o	-	-	+	o	o
Standardization	o	+ +	o	+	o	- -	- -	o
Marketability of available products	o	+ +	o	o	+	+ +	+ +	- -

Table 1. Subjective, Company-Specific Taxonomy Overview for Middleware Technologies

Therefore, we think an enterprise has to check for every application which middleware technology or combination of technologies suits best. Throughout the process of building a taxonomy the members of discussion get a feeling for the technologies. An enterprise should identify a key middleware technology as an *integration base*, which integrates applications using different middleware technologies. This key technology should be the technology which is presumably used for the most applications. E.g., some enterprises use CORBA as their key technology, because it offers platform and language independence and a mature component model. Others use an application server as their key technology because the majority of their applications are web-based e-commerce applications.

This outcome is obvious to the technology expert, but not for the manager. The ideal outcome for a manager would be, that one or a very limited set of products could be chosen. But at the enterprise scale it is not realistic that one product (or one specific product combination) serves best for all application requirements. A very detailed document describing the various properties, advantages, and disadvantages of technologies/products in detail is a good backdrop for specific discussion, but it gives a bad overview.

Therefore, we have added a rather simplistic overview table (similar to Table 1 – here we only summarize the technologies). The table is only meant as a starting point for discussions and for making/communicating a pre-selection. We use the following simple scale for rating of the taxonomy aspects: support for aspect is outstanding (+ +), support for aspect is good (+), support for aspect is available (o), support for aspect is not ready for the enterprise scale (-), and support for aspect is not or nearly not available (- -).

Note, that the simplicity of the table is provoking. And it is intended to be provoking, because this helps to start a discussion. E.g., an RPC user may heavily object, that RPC technologies are nearly not scalable. If the RPC user can argue for the technology and can argue that it is less work to build a bridge to the integration base than to use the integration base itself, the concrete application project, will probably use RPC. Afterwards, the taxonomy can be updated according to the experiences with that project.

Such considerations heavily depend on the company and the involved developers. If a department has several RPC experts and an RPC library framework, several of the aspects may require a quite different evaluation than in a company that has never used RPC before.

4 Integration and Coping with Change

So far, we have discussed how to make pre-selections for middleware technologies in very early stages of projects for single applications. One outcome was that no single technology can serve all requirements. Therefore, we need to come to a decision for the concrete application and we require an integration of (a) the technologies (and their services, tools, and processes) and (b) the involved (slightly) different paradigms. A special interest lies on the changeability at the technology seam, since it is a hot spot of the application.

4.1 Key Technology Decision and Integration Base

The software architecture is the first artifact in the development of a software system, that enables us to set priorities among competing concerns of the different stakeholders of the system. These concerns may be expressed in terms of quality attributes, as in [1], like performance, security, availability, modifiability, portability, reusability, integrateability, or testability. It is obvious that no architecture can maximize all of these quality attributes at once. The architect has to analyze the relevant requirements in terms of quality attributes. Influences for an architect are the architect's experience, the technical and organizational environment, and the stakeholders (like customer, end user, developing organization) of the system, who can be interviewed to find relevant scenarios with methods like SAAM [1] or [2]. The architect has to actively gather these information from the stakeholders by interviews and circulation of the results.

Our taxonomy does not lead to a distinct recommendation for one middleware solution. It just helps an architect of a concrete application to select an appropriate middleware solution from the possible alternatives. None of them is absolutely superior to other solutions, and unfortunately, each application demands different quality attributes. Therefore, the architects of any software system have to choose the appropriate solution for their application. This outcome is very unsatisfying regarding the aim to find an integration base technology/programming language at a very early stage of a project.

However, after understanding of the business cases for the software systems and elicitation/understanding of sufficient number of requirements (using scenario-based techniques if appropriate, sometimes other techniques, like formal requirement specifications, are necessary), an integration base can be identified. A sufficing number of applications is reached when the domain experts are sure that examples of most characteristic applications are investigated. One technology combination is chosen as an integration base. Concrete application developers are free in their choice of a technology, but the architecture must contain an interface, that conforms with the integration base. These interfaces are FACADES [5] that shield the application from direct access to the internals. The interfaces offer the application's services to client's based on different technologies.

Both integration base and concrete applications can be found by firstly performing a pre-selection of technologies, e.g., using the taxonomy. The result is a brief evaluation which technologies can not satisfy the requirements sufficiently. Afterwards larger, characteristic examples are investigated and candidate architectures for the examples using the different technologies are developed. These are evaluated for their architectural advantages/liabilities by comparison of the solutions and development/evaluation of (expected) change scenarios using software architecture analysis [1,8] (see Section 5 for an example).

The integration base is one kind of interface to which all applications offer their service and can comprise for instance an integrating technology and its component concept. To gain architectural flexibility in the integration base and in the software architectures of the applications the decision of each used technology should be performed stepwise on basis of change scenarios that are found and evaluated in interviews and circulation with the stakeholders. Figure 3 illustrates the various influences (ovals) and the derived artifacts (rectangles) in this process of finding a key technology. Dotted lines represent aspects which are evolving in distinct studies/implementations, while solid lines indicate continuously evolving aspects.

With the presented approach we give the application developers the freedom to develop applications with the technology that fits the application domain the best. And we have not ignored the subjectiveness in the technological decision process. Such ignorance makes developers feel uncomfortable and thus produces bad results. However, the developing department has to care for building a bridge to the integration base, if it is not available in the company already.

4.2 Object System Layer for Paradigm Integration

The open questions are, how to integrate another key technology with the integration base, how much efforts integration takes, and how much complexity the integration adds to an application system. In this domain we can present a quite concrete, technical solution, which relies on the component concepts from [6] and the Object System Layer architectural pattern [10].

We can see each subsystem as an opaque black-box, that is accessed via a Facade component. The Facade component includes a Facade object for the used middleware technology and wrapper objects that implement the calls of the Facade to the subsystem (if necessary). E.g., if we have a C legacy subsystem that uses RPC calls, we would extract a component with one distinct interface to the subsystem using RPC. Now we build an Adapter that is a second Facade to the subsystem and that has no other task than adaptation of calls using the integration base technology to the RPC interface.

In summary we add to each independent subsystem a component that explicitly defines the component's export interface. This interface is only way for other systems parts to access the subsystem and it is build with the middleware technology chosen for the subsystem. A simple Adapter integrates the integration base technology.

This simple approach lets us split up an existing system into self-contained subsystems and components. Thus we can build a component-oriented structuring for an existing legacy system in a piecemeal way. Therefore, our approach also provides a clear, piecemeal way for migration to the new middleware technology. In [7] we present a larger case study of such a migration process for a document archive system with the presented concepts.

Often paradigms of various programming languages and key technologies have to be integrated in a single component (especially in the Facade component of a subsystem). Often these models have to be integrated with concepts and languages that do not offer a notion of a certain paradigm at all, as the object-oriented paradigm that is introduced into languages, like C, by middleware, like CORBA, MOM, or TP monitors. For the enterprise context this integration of object/component concepts is especially important, since a large number of different models has to be integrated with the integration base technology/language.

This problem of integrating a foreign object system into a base technology/language can be solved by various approaches. We have documented the general underlying solution in the Object System Layer architectural pattern [10]. The solution builds or uses an object system as a language extension in the target language and then implements the design on top of this Object System Layer. It provides a well-defined interface to components that are non-object-oriented or implemented in other object systems. It makes these components accessible through the Object System Layer and then the components can be treated as black-boxes. The Object System Layer acts as a layer of indirection for applying changes centrally. There are several implementation variants of the Object System Layer pattern. Examples of popular Object System Layers are object-oriented scripting languages, libraries implementing an object-system, and object systems of key technologies.

Figure 4 shows the C/RPC subsystem with a CORBA integration base. Here, we must integrate the C procedural paradigm with the object system of CORBA. We propose to use an scripting languages, like XOTcl [11], as an Object System Layer to integrate with the integration base technology, because the flexible and dynamic language means of the scripting language allow us to easily integrate Adapters/Decorators at the Facade component. Rapid customizability of the Facade interfaces and the connection to the subsystem are important, because the interface section of black-box components are hot spots of distributed systems and the scripting language's component concept allows us to build compound components from several base language components. An (existing) Java/RMI subsystem can integrated with the C/RPC subsystem by giving it a similar CORBA Adapter to the RMI Facade.

Note, the similarity and symmetry of the scripting language solution with XOTcl and the Java solution. In both cases the Facade component has the task to serve as a component glue. This is the basic language design issue of scripting languages, like Tcl, which is designed as a glue for C or C++ components. Here, we re-build the same architecture in Java to have a glueing component that shields the subsystem. This way we have a clear stepwise way to migrate existing subsystems or subsystems build with another technology than the integration base into the enterprise's information system.

5 Distribution Center Example

In order to show how to apply the found results to concrete applications, we have concluded our study with several illustrative examples from various application fields of the enterprise. Here, we present one of these examples very briefly: a middleware solution of distribution centers for letters. A distribution center sorts letters by destinations. Various centers are connected to exchange letters and logistic data. Each center sorts letters for other centers. Inside of a center letters move around in standardized baskets. These baskets are sorted several times with sorting machines which are equipped with computers that are running non-standard operating systems. Non-standard letters have to be additionally sorted by human operators at a special sorting place. Before and after sorting the baskets are weight in order to control whether all letters have made it through the sorting machine or not. The balances for weighing the baskets are special peripheries which have to integrated into the system. Several NT and Unix workstations collect the data from the sorting process and compare the results. Furthermore various workstations are used for character recognition by human operators. Central computers are Leitstand, PPS, communication systems. They directly interact with central databases. The example can be seen in Figure 5.

First, we take a look at the requirements of the system. A middleware for the system should be capable of integrating the various platforms and languages used in the legacy systems. Since the system is a large, continuously working system, legacy applications have to be integrated and the migration to the middleware solution should be incremental. It should transparently encapsulate the special peripheries, like the balances. An important issue is scalability. On some days in the year (e.g., before Christmas) there is a considerable higher demand for sending letters than in other periods of the year. The system has to be integrated with other distribution centers and with the management's information systems for exchange of statistical data.

Now we map these requirements to our taxonomies' criteria. The system requires reliable network communication, programming language, and platform independability. An integrated component model should integrate various applications, legacy components, and special peripherals. The system would benefit from a transaction processing monitor or transaction service, because the database connectivity has to handle a larger number of transactions. Since the system is a very large system and should survive a long time, standardization is important in order to be independent of vendor politics. Finally the products must have proven a high reliability in an enterprise context, because failures in the system can cause considerable costs and severe damage to the image of the enterprise.

In the next step we make a pre-selection of candidate technologies. RPC approaches are not a superior solution, since their scalability properties are weak and they are poorly standardized. DCOM suffers from very limited platform independence and from its low-level scalability functionalities. Enterprise Java

Beans and RMI are not programming language independent, what makes it hard to incorporate existing legacy applications, their performance is weaker than in other approaches. For both DCOM and EJB/RMI it is questionable if the technologies are ready for the scale of the application. A transaction monitor alone has a weak component concept, weak language independability, and is not standardized. Mobile code systems would be the good paradigm for this application, because the code can move around with the baskets, each basket's procedure can easily be customized, mobile agents can gather statistical data locally and convey it to the management department by migrating to the management places, etc. But the current approaches do not seem to be ready for the scale of the application.

Three middleware technologies are candidates after the results of the initial discussion: CORBA, application servers, and MOM. Furthermore combinations are possible. CORBA meets most of the requirements: it is platform independent and language independent, offers an integrated component model, has high scalability and performance, is standardized, and the products are marketable. An application server has its weaknesses in programming language independence and probably in performance, if the application requires client-to-client interaction. In the given application example, most parts of the application are clients and servers at the same time. E.g., a PPS computer is a client to the sorting machines, when it gathers information, but it is a server in providing information about PPS decisions. If not every machine is wrapped behind its own application server, it is unclear how to cluster machines to application servers. The main drawback for message-oriented middleware in the given application is its missing standardization and the overhead of asynchrony, which is negative for net load and performance. It may result in further investments in stronger hardware.

After performing a comparison on the criteria of the taxonomy, we perform a software architecture analysis. For space reasons, here we just investigate three scenarios as examples in Table 2 very briefly. We find the relevance of the scenarios by interviews with the stakeholders. We assume that a change scenario that was often named is more likely to happen than one which is named only seldomly.

Overall we have built a list of all scenarios in the same style. Then we have given marks for each product in each scenario for evaluation. These were found by a discussion of the scenario descriptions with the stakeholders. Upon these marks we have made the concrete choice for the application. In the concrete example a CORBA based architecture was chosen, because it deals with most of the quality attributes better than its competitors, it is a good integration base for integration with other applications, and it handles most relevant change scenarios sufficiently. Nevertheless, for other applications or other enterprises other technologies or combinations could serve better.

Scenario Description	CORBA	Application Server	MOM
What happens if a new platform is introduced?	With CORBA it is quite easy to add a new platform support, if the platform is supported by any CORBA vendor, since the basic CORBA functionalities are quite compatible across vendor implementations.	The change depends on the support of the middleware vendor of the concrete product. With EJB servers it also depends on the availability of Java on the platform.	The change depends on the support of the MOM vendor of the concrete product. It may expensive and exotic platforms may not be supported.
What happens if licensing costs of a technology rise dramatically (e.g., through changes in vendor's pricing policy)?	Using CORBA the effort to change to another vendor depends to what extent vendor specific extensions/services are used. Generally a change is possible with foreseeable costs.	Application server specific application parts have to be re-written in order to change to a new vendor. Programming language depend parts normally, may be reused since for all prevalent languages (like Java, C, Tcl) several similar products are existing.	MOM products can cause significant problems in changing of the vendor, since their models are heavily vendor dependent and since there in not a great variety of comparable products at the market.
What happens if the database technology is changed?	All three technologies offer capabilities for encapsulation of legacy components/database wrappers. With a good design (a database access layer) it should be easy to change the database with all technologies. However, the abstraction from client interaction logic of a transaction monitor would serve better. Runtime means of adaptation (offered by some vendors) also help to transform one database wrapper to another.		
...	...	...	...

Table 2. Change Scenario Description

6 Related Work

The presented approach deals with the decision for key technologies. Firstly, a general overview and a subjective taxonomy are built. Then we mime example architectures, to find relevant scenarios. Finally, we find very concrete integration base technologies and explain concrete architectural solutions of legacy integration. Despite the earliness of the study, we provide quite concrete outcomes, without ignoring subjective experiences or company/application-specific aspects. There are several approaches, especially in the field of software architecture, that deal with parts of this process.

In [1] the software architecture analysis method (SAAM) is introduced. Several case studies are presented, including a case study of the CORBA architecture and the architecture of the web. Influences of software architecture on quality attributes and stakeholders of an organization are described in great detail. Other scenario-based architecture analysis methods are described in [2] and [8]. These approaches concentrate on the flexibility of architecture analysis in very early stages of software projects. This methods may be used as a part of our approach in order to find and evaluate scenarios. Generally, these approaches rather concentrate on more concrete architecture and not on very early evaluations and are, therefore, alone not suitable for an early key technology evaluation. They are not accompanied by a clear architectural vision for object-system or paradigm integration.

Architectural styles and patterns rather deal with last part of the process discussed in this work. In [15,4,14,1] architectural styles and patterns are discussed. In [12] the influence of the styles imposed by middleware technologies is investigated with the conclusion that no style serves best and, therefore, that different application have different middleware needs. Middleware induced styles should be made explicit in form of a style map, that can possibly be defined formally by a architecture description language. The case study in this work shows the entanglement of key technology decision and integration solutions. Therefore, these works are an important companion to the present work. With a clear knowledge of relevant styles and pattern it is easier to explain the integration of the technologies, object systems, and paradigms.

Brown discusses in [3] obstacles and important issues in applying and understanding middleware systems. He identifies a set of central aspects software managers have to adopt in order to successfully use and understand the benefits of middleware technology. Besides the aspects covered by our approach, for every application it has to be investigated, what the additional necessities for the usage of the technology with a concrete application are. Additional boundaries evolving with the usage of the technology have to be considered. Time and costs of adapting to the technology should accompany a final decision. These issues could be taken into concern in form of change scenarios. The approach is not as detailed as our approach and lacks a discussion of the integration problem.

A set of smaller studies solely provide comparison frameworks for middleware technologies (similar to our taxonomy). These approaches lack a discussion of the broader selection process and of the integration task. Raj [13] compares the middleware technologies CORBA, DCOM, and Java RMI very detailed at a implementational level. On a larger example he compares the benefits/liabilities by source code comparison. In [16] a detailed comparison of COM and CORBA technologies for distributed computing is given in form of a decision framework. From its intention it strongly resembles our taxonomy approach. Unfortunately, none of the named works gives a full fledged overview and a systematic comparison of all relevant middleware technologies.

Thompson [17] defines and positions middleware similarly to our work and propose a four step based approach to select a middleware technology. First the approach identifies the communication types within a business and between businesses. Then the underlying communication models of these types are classed into five communication models, which are conversational, request/reply, message passing, message queuing, and publish/subscribe. On the basis of these models middleware technologies are identified and finally evaluated on candidate architectures. The general steps are similar to our approach, but we doubt that the communication models are a detailed enough characterization of middleware technologies. Any organization of enterprise-size will most likely require all kinds of communication models. Most current middleware technologies implement more than one communication model. For both reasons the communication models can not serve as a good critierion for distinction. Moreover, in early studies when business requirements are not fully known, it can also be hard to find the relevant communication types.

7 Conclusion

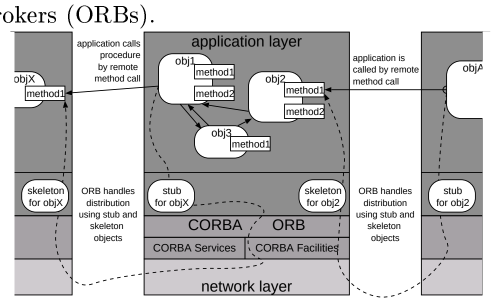
Key technologies, like middleware, have significant influence on the software architecture of an information system. The software architecture, in turn, has a severe impact on the realization of quality attributes of a company. Often evaluation studies on key technologies have to performed in early stages of projects for a large number of application. On the enterprise scale the application's requirements are in most cases too diverse to let an imposed, upfront key technology decisions over all applications seem sensible. An early study can identify the relevant requirements/quality attributes, map them in a taxonomy to the technology's properties, and identify an integration base. On example systems technology decisions and integration with existing system can be exemplified. With such a partial result that leaves a lot of freedom for the application designers, an early key technology study can make sense from the technological viewpoint. Rather simplistic overviews of the results, like taxonomies or change scenarios, serve as good starting points for discussion with non-experts in the technological field, if they are clearly mappable to their concerns. The best way to achieve such a mapping is an ongoing discussion with the stakeholders throughout the study.

References

1. L. Bass, P. Clement, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, USA, 1998.
2. P. Bengtsson and J. Bosch. Architecture level prediction of software maintenance. In *Proceedings of the International Conference of Software Engineering (ICSE99)*, Los Angeles, USA, 1999.
3. A. W. Brown. Mastering the middleware muddle. *IEEE Software*, 16(4), 1999.
4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-orinented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
6. M. Goedicke, G. Neumann, and U. Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00)*, Erfurt, Germany, Oct 2000.
7. M. Goedicke and U. Zdun. Piecemeal migrating of a document archive system with an architectural pattern language. to appear, 2000.
8. N. Lassing, D. Rijsenbrij, and H. van Vliet. Towards a broader view on software architecture analysis of flexibility. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, December 1999.
9. I. T. Ltd. The orbix architecture, 1993.
10. M.Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000.
11. G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.
12. E. D. Nitto and D. S. Rosenblum. On the role of style in selecting middleware and underware. In *Proceedings of the ICSE'99 Workshop on Engineering Distributed Objects*, Los Angeles, USA, 1999.
13. G. S. Raj. A detailed comparison of CORBA, DCOM and Java/RMI. http://www.execpc.com/ gopalan/misc/compare.html, 1998.
14. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.
15. M. Shaw. Some patterns for software architecture. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design 2*, pages 271–294. Addison-Wesley, 1996.
16. O. Tallman and J. B. Kain. COM versus CORBA: A decision framework. *Distributed Computing*, Sep-Dec 1998.
17. J. Thompson. Avoiding a middleware muddle. *IEEE Software*, 14(6), 1997.
18. S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneos environments. *IEEE Communications Magazine*, 14(2), 1997.

Common Request Broker Architecture (CORBA)

Description CORBA [18] is a distributed object system with the aim to realize distribution of object communication across different machines, vendors and software systems. Object method calls are invoked with the same principle as RPC from object stub (placeholder or proxy at the client side for an remote object on the server) to skeleton (interface implementation at the server side). Relevant functions are interface definition (with an IDL), localization and activation of distributed objects, communication among client and object, and distribution transparency. These issues are handled by object request brokers (ORBs).



Interoperability ORBs utilize the IIOP (internet inter-ORB protocol) in order to connect ORBs. The protocol on the TCP layer is designed to let all ORBs use the same protocol. The design of CORBA is generally aimed at language and platform independence. A common IDL lets components be specified through their interface without knowledge of internal implementation details. A disadvantage is that languages are broken down to a common denominator (partially solved by the data type any). The CORBA IDL is mapped to great variety of languages, like C, C++, Java, OLE, Eiffel, Smalltalk, etc. CORBA ORB implementations exists on nearly any commonly used platform.
Services Initially the CORBA messaging service and the event service were based on a simple push/pull model for message exchange through event primitives. Now the OMG specifies a more robust messaging service. Meanwhile several CORBA ORBs have implemented their own protocols or they can be combined with professional MOM products, like MessageQ, MQSeries, etc. The CORBA transaction service supports flat and nested transactions. Heterogeneous ORBs and (procedural) non-ORB applications can take part in a transaction. Some vendors even offer support for integration with commercial transaction monitors, like Tuxedo. The CORBA security services specification is one of the most detailed security specification existing, covering nearly every aspect of security, like integrity, authentication, access control, etc. It is achievable in three levels (0-2) from no to full security. Support for these services varies in different ORB implementations. The CORBA naming service enables to search for objects using their object name. It wraps several different traditional directory services. Some ORBs offer a fault-tolerant naming service.
Scalability ...
Performance ...
Standardization ...
Marketablility of the products ...

Figure 2. Technology Description: CORBA

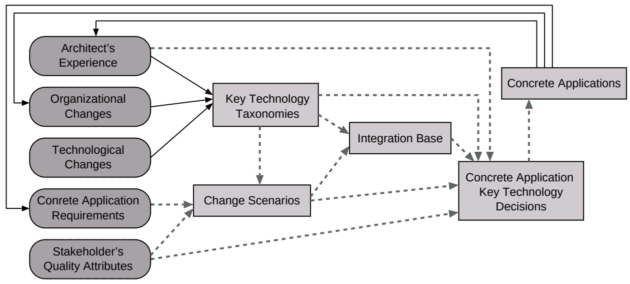


Figure 3. Influences and Artifacts in Key Technology Decision

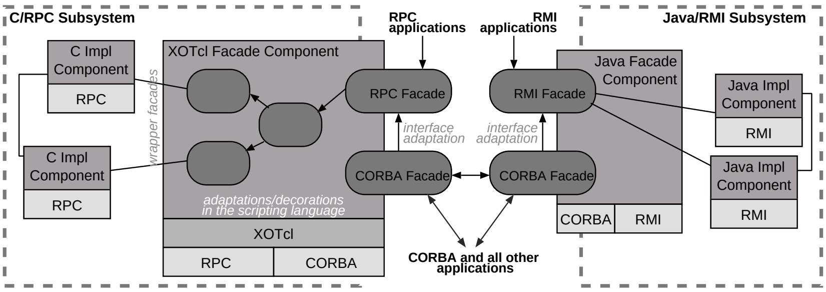


Figure 4. CORBA Integration Base with Java/RMI and C/RPC Subsystem

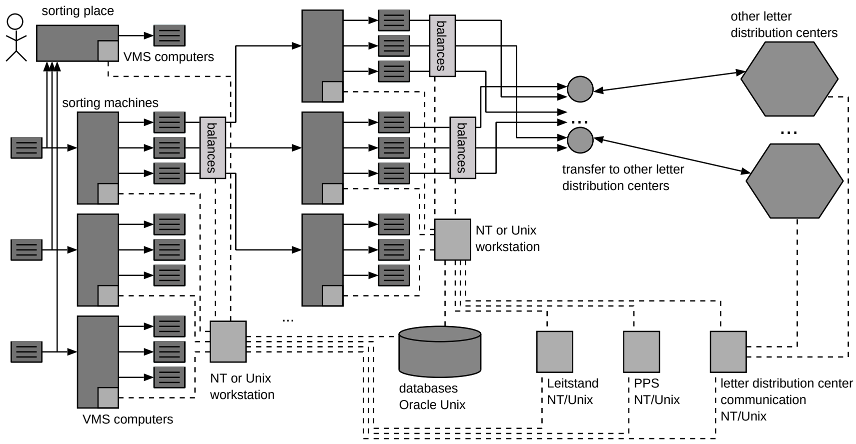


Figure 5. Letter Distribution Center Example